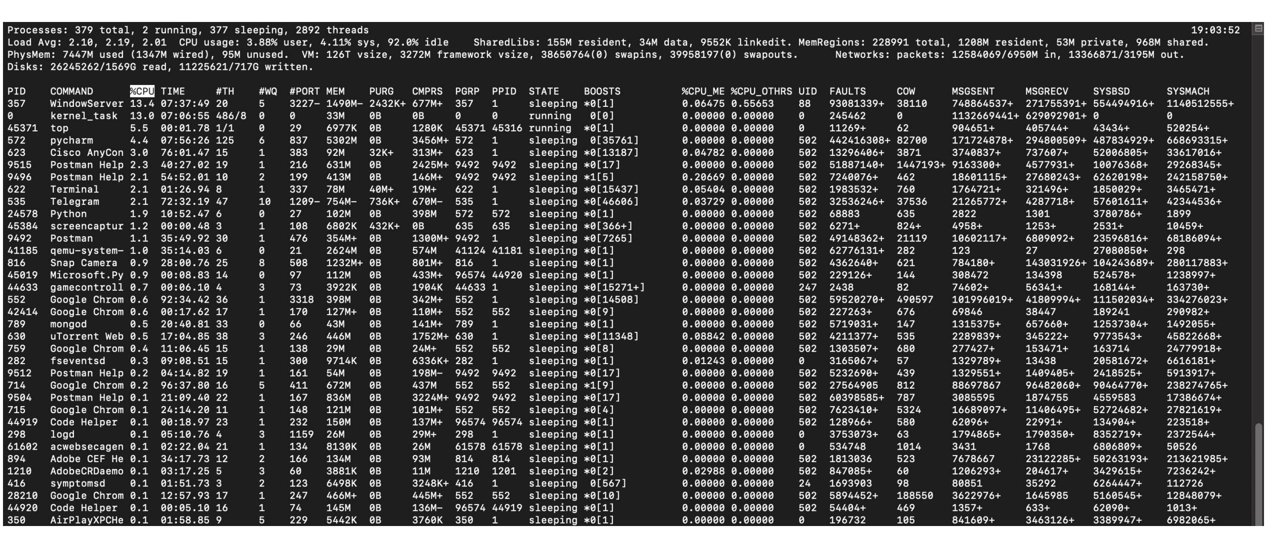
Процессы

Заумное определение процесса можно прочитать здесь.

Если говорить попроще — это некая сущность операционной системы, которая исполняет определенные инструкции.



На картинке вы можете наблюдать вызов linux-команды **top**, где перечислены все запущенные на системе процессы и сопутствующая с ними информация.

Заголовок первой колонки — **PID**. Это уникальный id процесса (process identifier).

Давайте запустим программу с несколькими процессами. Библиотека для работы с ними называется multiprocessing.

```
from multiprocessing import Process
import os


def info():
    print("Имя модуля:", __name__)
    print("Родительский PID процесса:", os.getppid())
    print("PID дочернего процесса:", os.getpid())


def f():
    info()
    pass


if __name__ == "__main__":
    info()
    p = Process(target=f)
    p.start()
    p.join()
```

```
Имя модуля: __main__
Родительский PID процесса: 572
PID дочернего процесса: 28252
Имя модуля: __mp_main__
Родительский PID процесса: 28252
PID дочернего процесса: 28254
```

Из процесса мы можем получить id текущего процесса и PID родителького процесса (PPID): процесса, который породил текущий, используя системный вызов.

В данном примере PID 572 — это процесс Pychram, который породил подпроцесс, запустив код выше.

Порождение процессов

Библиотека multiprocessing поддерживает 3 способа порождения процессов:

Spawn. Запуск нового процесса интерпретора python. Будут унаследованы только те ресурсы, которые необходимы для запуска процесса.

Fork. Системный вызов fork. Полное копирование дочернего процесса.

Forkserver. Инициализация однопоточного процесса сервера. При необходимости порождения дополнительных процессов происходит запрос к серверу, и он, в свою очередь, создает дополнительный процесс.

В следующих главах многопроцессорность в python нам не понадобится. Поэтому подробно останавливаться на режимах создания дочерних процессов мы не будем.

Разделяемая память

Определение

При инициализации процесса выделяется оперативная память. Процессы не имеют доступа к ресурсам других процессов без специальных механизмов обмена данными. Убедимся в этом на примере.

```
import time
from multiprocessing import Process
import os

data = dict()


def info():
    print("PID процесса:", os.getpid())
    print("data:", data)


def p1():
    data["p1"] = True
    info()


def p2():
    data["p2"] = True
    info()


if __name__ == "__main__":
    info()

    proc_1 = Process(target=p1)
    proc_1.start()

    proc_2 = Process(target=p2)
    proc_2.start()

    proc_1.join()
    proc_2.join()

    info()
```

```
PID процесса: 29871
data: {}
PID процесса: 29874
data: {'p2': True}
PID процесса: 29873
data: {'p1': True}
PID процесса: 29871
data: {}
```

Мы видим, что ни один из трех процессов (1 родительского и 2 дочерних) ничего не знает об изменениях содержимого словаря. Это обсуловлено тем, что переменные хранятся в оперативной памяти, выделенной для процесса. Память процессов изолирована, поэтому содержимое словаря в каждом процессе разное.

С механизмами обмена данными между процессами можно ознакомиться тут.

CPU операции с применением multiprocessing

Убедимся, что в случае с несколькими процессами CPU-операции действительно будут отрабатывать быстрее. Запустим сначала синхронную программу, замерим время и сравним с теми же вычислениями, но распараллеленными на несколько процессов.

Синхронное вычисление в одном процессе:

```
import time


def countdown(id: int):
    i = 0
    begin = time.time()
    while i < 50_000_000:
        i += 1

    print(f"ID: {id} DONE duration: {time.time() - begin}")


COUNT = 2
begin = time.time()
print("START calculation")
for i in range(COUNT):
    countdown(i + 1)

print(f"FINISH calculation duration: {time.time() - begin}")
```

```
START calculation
ID: 1 DONE duration: 1.199383020401001
ID: 2 DONE duration: 1.1890840530395508
FINISH calculation duration: 2.3886091709136963
```

Математика в разных процессах:

```
import multiprocessing
import time


def countdown():
    i = 0
    begin = time.time()
    while i < 50_000_000:
        i += 1

    return f"ID: {multiprocessing.current_process().name} DONE duration: {time.time() - begin}"


def test():
    PROCESSES_COUNT = 2
    begin = time.time()

    print("START calculation")
    with multiprocessing.Pool(PROCESSES_COUNT) as pool:
        results = list()
        for _ in range(PROCESSES_COUNT):
            results.append(pool.apply_async(countdown))

        for r in results:
            print(r.get())

    print(f"FINISH calculation duration: {time.time() - begin}")


if __name__ == "__main__":
    multiprocessing.freeze_support()
    test()
```

```
START calculation
ID: SpawnPoolWorker-2 DONE duration: 1.2518351078033447
ID: SpawnPoolWorker-1 DONE duration: 1.2517151832580566
FINISH calculation duration: 1.336179256439209
```

Легко заметить, что в этом случае использование нескольких процессов оказалось быстрее, нежели простые синхронные выполнения. Разницу можно почувствовать на длительных вычислениях, так как системные вызовы порождения процессов и их уничтожения тоже занимают время.

Минусы многопроцессного программирования

1. Накладные расходы на инициализацию дочернего процесса. В зависимости от способа инициализации нужно выделить ресурсы в системе и скопировать все, что было в родительском процессе.
2. Раздельная память. Нет возможности напрямую обмениваться данными между процессами.
3. Ограничение системы на количество процессов и выделенное им процессорное время.
4. Накладные ресурсы на системные вызовы переключения процессов.
5. Информация, передаваемая непреднамеренно завершенным процессом, может быть повреждена. Например, если мы решили убить процесс, или он был завершен из-за возникшего исключения.

Особенности работы при многопроцессном программировании

1. Избегайте перемещения больших объемов данных между процессами.
   Если это необходимо, для взаимодействия между процессами используйте queues или pipes..
2. Не забывайте убивать «зомби»-процессы.
   Нужно вызывать метод **join()** всех отработавших процессов. Если этого не сделать, процесс повиснет в системе.
3. Аргументы функций должны быть **pickled.** pickled-аргументы перечислены тут.
4. Если требуется передавать данные между процессами, программа должна быть спроектирована так:
   — родительский процесс передает данные в очередь, т.е. является единственным писателем информации
   — дочерние процессы получают данные для обработки, т.е. являются читателями / потребителями информации.
   Пример. Родительский процесс добавляет в очередь задачи и параметры, которые нужно решить, а дочерние процессы по мере освобождения получают эту информацию и производят работу по ее обработке.

В этом уроке мы не освещаем обмен данными и синхронизацию между процессами, но вы можете самостоятельно изучить официальную документацию:

- обмен данными
- синхронизация между процессами